Using Machine Learning to Infer Constraints for Product Lines

Paul Temple, José Angel Galindo Duarte, Mathieu Acher, Jean-Marc Jézéquel

Using Machine Learning to Infer Constraints for Product Lines

Paul Temple
University of Rennes 1, France
paul.temple@irisa.fr

José A. Galindo
Inria, Rennes, France
jagalindo@inria.fr

Mathieu Acher
University of Rennes 1, France
mathieu.acher@irisa.fr

Jean-Marc Jézéquel
University of Rennes 1, France
jezequel@irisa.fr

ABSTRACT

Variability intensive systems may include several thousand features allowing for an enormous number of possible configurations, including wrong ones (e.g. the derived product does not compile). For years, engineers have been using *constraints* to a priori restrict the space of possible configurations, i.e. to exclude configurations that would violate these constraints. The challenge is to find the set of constraints that would be both precise (allow all correct configurations) and complete (never allow a wrong configuration with respect to some oracle). In this paper, we propose the use of a machine learning approach to infer such product-line constraints from an oracle that is able to assess whether a given product is correct. We propose to randomly generate products from the product line, keeping for each of them its resolution model. Then we classify these products according to the oracle, and use their resolution models to infer cross-tree constraints over the product-line. We validate our approach on a product-line video generator, using a simple computer vision algorithm as an oracle. We show that an interesting set of cross-tree constraint can be generated, with reasonable precision and recall.

Keywords

software product lines; machine learning; constraints and variability mining; software testing; variability modeling

1. INTRODUCTION

Variability intensive systems, such as the Linux kernel, may include several thousands of features allowing for an enormous number of possible configurations. Among them, a lot of theoretically possible configurations are, however, not valid for a specific task (*e.g.,* the kernel does not compile). This may happen due to some hidden constraints (mutually exclusive features, incompatible values for constants, *etc.*). It is then extremely painful for the engineer to discover, late in the process (at compilation time or even worse at testing time), that her carefully chosen set of features is actually invalid.

The space of actually possible products for a specific task can be seen as a subspace of the initial space of possible products given by a variability model. A typical way to go from the initial space to the desired subspace is to add constraints to the variability model to restrict the initial space down to the desired subspace. It can be equally seen as a way to constrain the acceptable inputs of the variables of an automatic product derivator.

The challenge addressed in this paper is to try to automatically synthesize a set of constraints that would be both precise (allow all correct configurations) and complete (never allow a wrong configuration with respect to some oracle). We propose the use of machine learning to infer such product-line constraints from an oracle that is able to assess whether a given product is correct (*e.g.,* the compiler in the case of the Linux kernel).

We propose to randomly generate products from the product line, keeping for each of them its resolution model. Then we classify these products according to the oracle, and use their resolution models to discover combinations of features and/or range of values that make a configuration invalid according to the oracle. We, then, turn these into cross-tree constraints over the product-line.

We validate our approach on a product-line video generator developed in the industry [1,12], using a simple computer vision algorithm as an oracle.

The rest of the paper is organized as follows. Section 2 presents the overall principle of our approach and discusses the kind of product-lines it is suited for. Section 3 applies our method on an existing video generator. Section 4 presents the experimental results in terms of meaningfulness, precision and recall and discusses threats to validity. Section 5 indicates possible directions of improvements. Section 6 discusses related works. Section 7 concludes with perspectives open by this work.

2. METHOD

We now describe a general method in which we leverage machine learning techniques to infer the constraints of a product line.

Figure 1 describes the process we followed up. We assume that there is a variability model that documents the configuration options of the product line under consideration.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

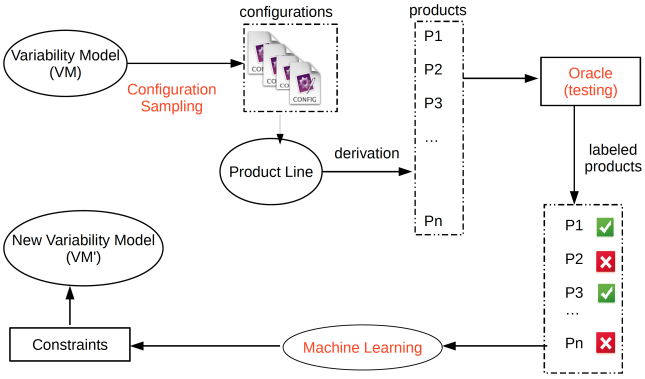


Figure 1: Sampling, testing, learning: process for inferring constraints of product lines

In such variability model, options can be Boolean features or numerical attributes. From a configuration (see Definition 1), product line artifacts are assembled and parameters are set to derive a product. The variability model may already contain some *constraints* that restrict the possible values of options (*e.g.,* the inclusion of a Boolean feature implies the setting of a numerical value).

Definition 1 (Configurations, variability model). *A* configuration *is an assignment of values for all options of a variability model. A configuration is valid if values conform to constraints (*e.g., *cross-tree constraints over attributes). A variability model VM characterizes a set of valid configurations denoted $[\![VM]\!]$.*

A first step in the process is *configuration sampling*. It consists in producing valid configurations (see Definition 1) of the original variability model VM. The set of sampled configurations is a subset of $[\![VM]\!]$. Numerous strategies can be considered such as the generation of T-wise configurations, random configurations, *etc.* [3, 8, 12, 16, 18, 22, 23, 27]

Second, an *oracle* is reused or developed. It tests the validity of the derived products corresponding to configurations. The notion of validity is specific to a product line and a usage context. It may refer to the fact the product does compile, does not crash at run-time, passes the test suite, and/or does meet a particular quality of service. In the reminder, we use the term *"faulty" configuration* for referring to such irrelevant products (see Definition 2). An oracle is used to create two classes of configurations: It labels as either faulty or "non faulty" the configurations in the sampling.

Definition 2 (Oracle and faulty configuration). *An oracle is a function that takes a derived product as input and returns true or false. A faulty configuration is a configuration for which the oracle returns false when taking as input the corresponding derived product.*

A third step is to use a *machine learning* procedure that operates on the labeled configurations and automatically infers constraints. A new variability model VM' is created by adding the newly identified constraints to the original variability model. Therefore, the new variability model VM' is a specialization [33] of VM and $[\![VM']\!] \subset [\![VM]\!]$. In other words, VM' forbids faulty configurations that were initially considered as valid in VM.

Instead of using machine learning, an alternate and sound approach is to remove faulty configurations from the original variability model. It consists in negating all faulty configurations and then making their conjunctions (see Definition 3). However, this approach is very limited since (1) it only removes a typically small number of configurations – only those that have been sampled and tested; (2) it does not identify which configuration options and values are involved and the root causes of the fault.

Definition 3 (Sound approach). *Given a set of faulty configurations $\{fc_1, fc_2, ..., fc_n\}$ a sound approach computes a new variability model VM' such that $VM' = VM \wedge \epsilon$ where $\epsilon = \bigwedge_{i=1..n} \neg fc_n$*

A simple removal of faulty configurations is thus not a viable solution for product lines exhibiting a large number of configuration options or numerical values. As an oversimplified example, let say we have faulty configurations $\{fc_1, fc_2, fc_3, \ldots, fc_n\}$. Among values of attributes and features, the attribute A varies as follows: $A = 0.265$ in fc_1, $A = 0.26$ in fc_2, $A = 0.275$ in fc_3, ..., $A = 0.29$ in fc_n. With a basic case by case extraction, we cannot infer that (perhaps) $0.26 < A < 0.3$. The use of machine learning can improve the inference of constraints through the prediction of ranges of values that make configurations faulty.

The expected benefit is to discard much more faulty configurations with the inference of constraints: Figure 2 summarizes the potential of machine learning. A rectangle is used to represent $[\![VM]\!]$. The set of configurations that can be detected as faulty is represented as red clouds/rectangles in Figure 2. The precise set is a priori unknown; this is precisely the problem.

Faulty configurations detected by an oracle, are included in this set (see crosses in Figure 2). The enumeration and testing of configurations for covering the whole set of faulty configurations will take a large amount of resources and time. In response, machine learning can infer a set of constraints delimiting *sets* of faulty configurations (instead of only forbidding individual configurations). Thanks to learning and prediction, we expect the capture of additional faulty configurations *without the cost of testing those configurations.*

The ideal case is that machine learning accurately classifies configurations as faulty, including configurations that were not part of the sampling. We represented that as the dashed rectangle exactly corresponding to the red rectangle (see left-hand side Figure 2). However, machine learning might produce false positives. That is, some configurations are classified as faulty whereas they are actually valid from the oracle's perspective. An example is given in Figure 2 with the red cloud and dashed rectangle at the bottom: some configurations are included outside the red cloud and in the blank area. Another kind of misclassification can happen when the dashed rectangle is included in the red cloud (see right-hand side of Figure 2). In this case, machine learning fails to classify some configurations as faulty and is incomplete. Despite specific cases in which machine learning can be unsound and incomplete, we expect that, in general, learning constraints enables to capture more faulty configurations than a simple conjunction of negated configurations.

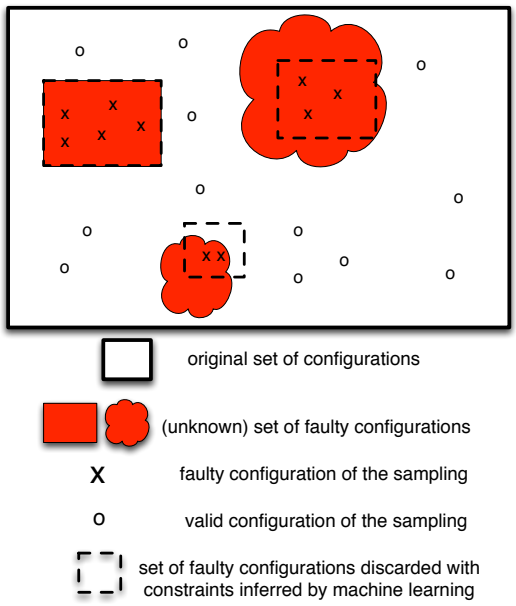


Figure 2: Constraining the configuration space

3. CASE STUDY

In this section we apply and evaluate the proposed method with a real-world product line, i.e., MOTIV which is a highly-configurable video generator developed in the industry [1]. Our objective is to address the following research questions:
RQ1) *Do extracted constraints make sense for a computer vision expert?*
RQ2) *What is the precision and recall of our learning approach?*
RQ3) *What are the strengths and weaknesses of our approach compared to existing techniques?*

3.1 Case and Problem

Given a configuration file, the MOTIV generator synthesizes a video variant with specific properties (luminance, trajectory of vehicles, noises, distractors, etc.). The software generator is written in Lua and implements numerous complex, parameterized transformations for synthesizing variants of videos [1, 12].

The motivation behind the generator is that the current practice for benchmarking tracking algorithms (*e.g.,* algorithms that track objects of interests in a scene) is limited to a *manual* collection of video sequences. Such manual effort is error-prone and time-consuming since it requires to (1) shoot video sequences in real environments and (2) annotate videos with a *"ground truth"*. As a result, benchmarks with limited size and diversity are usually employed. In response, this MOTIV generator has been developed to *automatically* produce a large and diverse set of video sequences.

The generator comes with a variability model (an excerpt is shown in Fig. 3). The model organizes the features and attributes in a hierarchical tree. Some features are Boolean (*i.e.,* included or not) while others, called attributes, are defined over continuous ranges of numerical values (see bottom of Figure 3). A given configuration is obtained by choosing a particular value for all these features and attributes. The software generator finally exploits the selected values to produce a variant corresponding to a configuration. Users can control the generation process to cover a large diversity of video variants and thus challenge tracking algorithms under different varying setups.

Problem. Users quickly noticed that the specification of constraints in the variability model is crucial for the video generator. Without constraints, many configurations lead to the generation of unrealistic video variants, due to the incompatibility between features and attributes' values. Beyond usability problems, this is an issue for two major reasons. First, the production of videos has a cost (about half an hour of computation on average per video variant). As a result, the synthesis of large datasets with thousands of video variants (as originally planned by industrials) would exhibit a lot of irrelevant videos, thus wasting computation power. Second, tracking algorithms performed on the synthesized videos are computationally expensive, which, in case of irrelevant videos, is again a waste of time and resources. Our early effort [1, 12] for properly formalizing the variability was thus not sufficient; we need to enforce the generator with constraints to make it usable and useful for practitioners.

Although some basic constraints have been manually specified, the generator still produced irrelevant video variants. In order to capture additional constraints and gather some knowledge, we have made several iterations with the developers of the video generator through meetings and mail exchanges. Finally, we came to the conclusion that either an analysis of the Lua source code or a further effort for manually specifying constraints presents strong limitations. It is mainly due to the fact that (1) the configuration space is extremely large (see hereafter for more details); (2) there is not enough knowledge to comprehensively capture constraints over features and attributes' values.

A manual exploration of the configuration space requires substantial efforts for both setting configuration values, assessing the quality of the output (videos), and learning from defects. We thus propose to use the method we described in the previous section to automate all these tasks, including the inference of constraints with machine learning.

We now detail how we instantiated every part of our method (sampling, testing, learning) within our case study.

3.2 Solution for Inferring Constraints

Figure 4 presents the entire process of extracting constraints of the video generator. A set of configuration files is first sampled from the variability model; it acts as a training set. The Lua generator derives a video variant for each configuration. An oracle is then used to label videos as faulty or non faulty by computing the quality of each video. Finally, a machine learning process is executed to extract the constraints and re-inject them into the original variability model. We now detail each step of the process.

3.2.1 Generating a training set out of the variability model

In the MOTIV case study, the variability model exhibits numerous features and attributes whose range of values are reals and continuous. Figure 3 presents an excerpt of its hierarchical structure and possible values for features and attributes. In total, the variability model contains: 42 real attributes, 46 integer attributes, 20 Boolean features, and

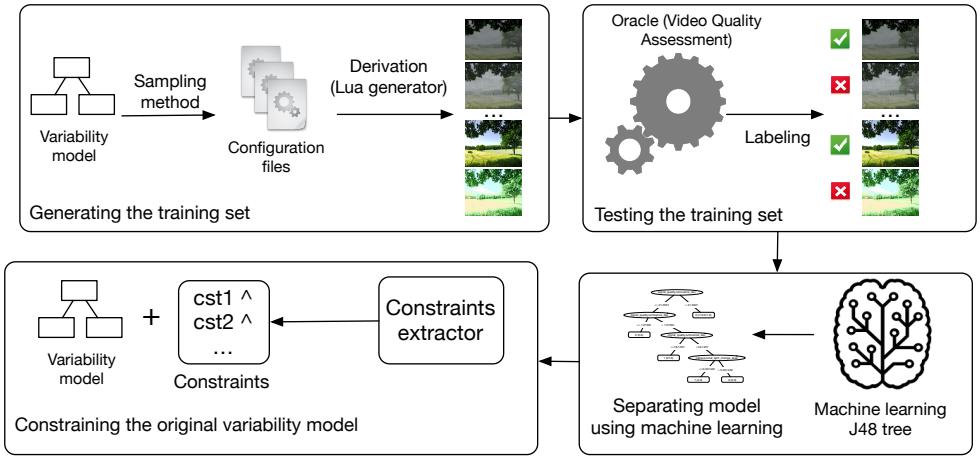


Figure 4: Learning method on the video generator

140 constraints (mainly constraints specified by the experts). The ranges vary between 0 and 6 for the integers domains, and in average between zero and 27.64 for the real domains with a precision of 10^{-5}. This would end up in approximately a total of 10^{103} configurations (not considering constraints): 2^{20} (because of the boolean variables) $\times 6^{46}$ (because of the integer variables and) $\times 2764000^{42}$ (because of the real variables). Overall the variability model presents the particularities of encoding a large configuration set with lots of non-Boolean values.

The nature of this model has encouraged us to develop a new version of the FAMILIAR tool suite [2, 12]. We implemented a solution capable of natively coping with real attributes which rely on the Ibea solver[1] provided with Choco 3 [26].

To generate a training set for the machine learning process, we need to produce a set of valid configurations. Different sampling techniques [3, 8, 12, 16, 18, 22, 23, 27] can be considered but some of them are not applicable to our case since we have to deal with real and integer values. We implemented the following procedure. First we randomly pick a value for each attribute within the boundaries of its domain. Then, we propagate the attribute values to other values with a solver; the goal is to avoid invalid values. We continue until having a complete and valid configuration. We reiterate the process for collecting a sample of configurations.

3.2.2 Oracle

Some videos of the generator are not of interest for computer vision algorithms and humans. Typically, these are videos in which the vision system cannot perceive anything or cannot distinguish moving objects from other ones (*e.g.,* distractors). Image Quality Assessment (IQA) typically tries to understand the conditions under which the vision system is likely to fail this kind of distinction. We implemented an IQA oracle, presented in [11], that can automatically assess whether a video is faulty. The principle is to perform a Fourier transformation and to reason about the resulting distribution of Fourier frequencies. Such a technique evaluates the quality of a single image. To reduce the time and cost of the oracle, we sample a video into a smaller set of images. After applying the IQA method on sampled images, we aggregate results to decide whether a video is faulty or not. We empirically set a threshold: If, at least, half of the images are declared faulty, then the whole video sequence is considered faulty.

3.2.3 Machine Learning (ML)

Using our oracle, we assigned a faulty or non faulty label for each video of the sample. We use a *Machine Learning (ML)* algorithm to understand the relationship between faultiness of videos and features/attributes' values. Different ML methods exist. Some of them are sophisticated and perform only a few classification errors while others are less advanced but are much more understandable when vizualizing the output model. In our case, we wanted to obtain a high level of understandability when extracting constraints. Specifically, we employed *Binary Decision Trees*.

Figure 5 presents an excerpt of the decision tree obtained from the Weka[2] software. In this tree:

- *ovals* represent features (written inside) on which decisions will have to be taken;
- *labels over edges* represent threshold value to decide which path to take;
- *rectangles* represent leaves of the tree and groups of configurations that are mainly of the same class.

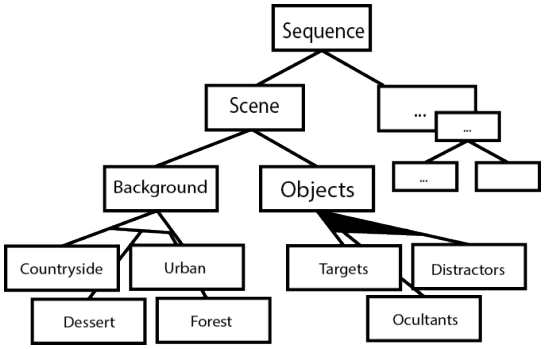
Leaves present several pieces of information. First, there is the label of the most represented class. In our case it is either '1' (faulty) or '0' (non faulty). Second, the number of configurations associated to the label. Finally, the number of configurations of the other class. These are classification errors, typically configurations that are at the boundary of two classes.

As an example, configurations having a value higher than 21..3521 set for feature "signal_quality.luminance_dev" will reach the leaf in the top right corner of Figure 5. It means that 110 configurations (out of 147) are labelled '0' and one is classified as faulty.

3.2.4 Extracting constraints

[1] http://www.ibex-lib.org/

[2] http://www.cs.waikato.ac.nz/ml/weka/




```
1  //Distractors
2  real distractors.butterfly_level [0.0 .. 1.0]
3  real distractors.bird_level [0.0 .. 1.0]
4  real distractors.far_moving_vegetation [0.0 .. 1.
       0]
5  real distractors.close_moving_vegetation [0.0 ..
       1.0]
6  real distractors.light_reflection [0.0 .. 1.0]
7  real distractors.blinking_light [0.0 .. 1.0]
8  //Capturing conditions
9  real camera.vibration [0.0 .. 1.0]
10 real camera.focal_change [0.0 .. 1.0]
11 real camera.pan_motion [0.0 .. 1.0]
12 real camera.tilt_motion [0.0 .. 1.0]
13 real camera.altitude [0.0 .. 5.0]
14 real capture.illumination_level [0.0 .. 1.0]
15 // Signal quality
16 int signal_quality.force_balance [0 .. 1]
17 real signal_quality.luminance_mean [0.0 .. 255.0]
18 real signal_quality.luminance_dev [0.0 .. 255.0]
```

Figure 3: Variability model excerpt of the generator

Once the decision model has been created, we traverse the tree and reach every leaf. Are remembered only leaves labeled '1' meaning their path is extracted. We consider that a path is a set of decisions, where each decision corresponds to the value of a single feature. We create new constraints by building the negation of the conjunction of the different decisions to make along the path to reach a faulty leaf. In case features are repeated, some simplifications are performed. In the example of Figure 5, we can extract the two following simplified constraints:

```
1  !(signal_quality.luminance_dev > 1.01561 &&
       signal_quality.luminance_dev <= 18.1437)
2  !(signal_quality.luminance_dev <= 21.3521 &&
       signal_quality.luminance_dev > 18.1437 &&
       capture.local_light_change_level <= 0.481449
       )
```

4. EXPERIMENTS

4.1 Experimental Setup

To generate a training set, we sampled 500 configuration files from the MOTIV variability model. All of them are given to the video generator to create 20 seconds long videos. The process of deriving associated video variants takes about 30 minutes in average per video. As we have to create numerous videos, we used a cloud-based architecture for distributing the computations. To decrease the influence of randomly creating training set (which could result in advantageous or disadvantageous settings), we run the experiment of learning and validating results 20 times (see Section 4.2.2 for more details). After the synthesis of videos, the oracle we presented in Section 3.2.2 assigned "non faulty" or "faulty" labels to videos. In total, the oracle labelled 53 videos as faulty on average, *i.e.,* roughly 10% of the videos.

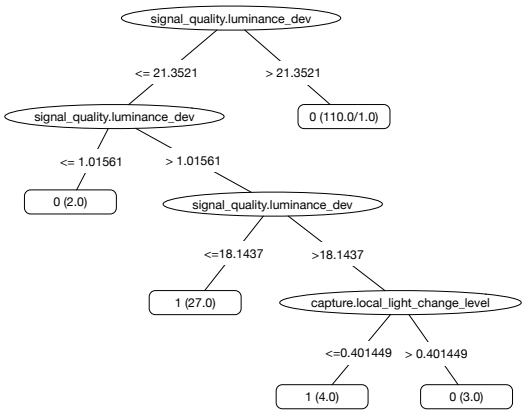


Figure 5: An excerpt of the decision tree built from a sample of 500 configurations/videos

We used decision trees with Weka to perform the ML part. Weka offers different implementations of Binary Decision Trees. We used J48 since it is the most widely used. Various options can be tuned to increase the classification performances. This process of selecting the best set of parameters is application-dependent, so we used the default ones set by Weka. The reader will find all the experimentation data at http://learningconstraints.github.io.

4.2 Results

4.2.1 *RQ1) Do extracted constraints make sense for a computer vision expert?*

The first research question focuses on the readability and comprehensibility of extracted constraints from an expert point of view. To be able to answer our question, we asked to a computer vision expert and advanced user of the video generator whether the extracted constraints did make sense or not. The expert told us that constraints are globally understandable and actually help understanding interactions that can occur between features/attributes. Importantly, he noted that constraints are in line with the definition of the oracle: Some combinations of values can indeed participate to the degradation of the perception of video contents.

Specifically, Figure 6 shows a short constraint only constituted by two terms. This constraint puts together two images quality criterion (blur and static noise) that can indeed degrade the quality of videos. In Figure 7, the constraint involves other features that have an effect on the quality of videos: compression and illumination. Interestingly, blur is present again. In Figure 8, blur, compression, and dynamic noise are features that are related to quality criterion of videos as well. Overall, all features/attributes previously mentioned make sense with regards to the oracle we implemented (see Section 3.2.2). Too much noise, poor illumination and blurs: all these factors can indeed degrade the quality of videos and produce the kind of videos our oracle is able to detect as faulty.

In general, extracted constraints make sense and the an-

```
!(signal_quality.blur_level > 0 && signal_quality
    .static_noise_level <=0.135519)
```

Figure 6: A constraint extracted from our case study

```
!(signal_quality.blur_level < 0 && signal_quality
    .compression_artefact_level > 0.363438 &&
    capture.illumination_level <= 0.609669 &&
    signal_quality.compression_artefact_level>0.
    436673 && vehicle5.trajectory >6 && vehicle1
    .identifier <=11)
```

Figure 7: A constraint extracted from our case study

swer to RQ1 is positive. However there is room for improvement (see also Section 5). Specifically the expert complains about the presence of features/attributes that are not relevant and disturb the reading. For instance, in Figure 7, $vehicle1.identifier <= 11$ does not make much sense. Indeed, the kind of vehicles should not have any influence on the definition of faulty videos. The expert has to somehow ignore this kind of information.

4.2.2 *RQ2) What is the precision and recall of our ML approach?*

In the rest of this section, we consider that a ML approach computes a new variability model denoted VM' such that $VM' = VM \wedge \Delta$ where Δ is a conjunction of inferred constraints.

The overall classification performance of ML is not perfect, *i.e.,* 90.56% on average after training the decision tree on 500 configurations/videos. This is due to the fact that despite there is a perfect knowledge over training data, ML tries to avoid over-fitting to be able to generalize what have been learnt. To have a better idea of the number of errors that our approach is likely to perform, we tested the output classification model while producing a new data set with configurations that were not in the 500 original configurations.

To do that, we generated another set of 4000 configurations and videos. We used again a cloud-based infrastructure to synthesize these variants. Our oracle labelled every video of this new data set. It resulted in 370 faulty videos on average. Then we compared the decision of the oracle with the decision of the variability model augmented with extracted constraints. We run the experiment 20 times by randomly changing the training set (500 configurations) as well as the validation set (4000 configurations).

In particular, we are interested to know whether a configuration labelled as faulty with the oracle is now forbidden by VM'. This comparison is usually performed through a

```
!(signal_quality.blur_level <= 0 &&
    signal_quality.compression_artefact_level <=
     0.363438 && signal_quality.
    dynamic_noise_level<=0.428141 &&
    signal_quality.force_balance=0 && capture.
    illumination_level <= 0.12151)
```

Figure 8: Another constraint extracted from our case study

		Oracle	
		Faulty	Non-faulty
variability model (VM')	Faulty (invalid)	234 (±57.899)	69.5 (±26.973)
	Non-faulty (valid)	141.1 (±60.440)	3566.2 (±25.804)

Table 1: Confusion matrix of our experiment

confusion matrix presented in Table 1.

In this table, columns are the labels given by the oracle and rows are labels given by our variability model. Cells present the average of classification over 20 runs as well as standard deviation (under brackets). The main diagonal of this matrix tells us where the two labels agree. The other diagonal provides classification errors of our variability model compared to the oracle:

False Positives (FP) are configurations "faulty"[3] in VM' whereas they are classified as non-faulty by the oracle. The ML approach has inferred too restrictive constraints that now uselessly forbid "non faulty" configurations.

False Negatives (FN) are configurations "non faulty" in VM' whereas they are classified as faulty by the oracle. The ML approach fails to infer constraints that could have forbidden "faulty" configurations.

Precision is the measurement assessing the number of correct classifications performed for a class (main diagonal) regarding the total number of classification made for this class (sum of a row). Over the 20 runs, the mean precision for the class "non-faulty" is : $P_{mean-non-faulty} = \frac{3566.2}{3566.2+141.1} \simeq 0.96$. Similarly, precision for the class "faulty" is $P_{mean-faulty} \simeq 0.77$.

The overall precision is the mean of the two values: $P_{overall} = \frac{0.96+0.77}{2} = 0.865$, *i.e.,* the classification will roughly perform well 9 times out of 10.

Recall is the measurement assessing the number of correct classification performed for a class regarding the total number of configurations declared by the oracle for this class (sum of a column). Similarly to the precision, recall can be computed for each class and then combined into an overall measure. This gives : $R_{mean-non-faulty} = \frac{3566.2}{3566.2+69.5} \simeq 0.98$; $R_{mean-faulty} \simeq 0.62$ and $R_{overall} = \frac{0.98+0.62}{2} = 0.80$. Here, recall is lower for the "faulty" class which gives us an idea of how difficult it is to understand the distribution of this class. This difficulty can come from the fact that there are fewer examples in the class "faulty" than in the class "non faulty".

4.2.3 *RQ3) What are the strengths and weaknesses of our approach?*

We now compare the properties of a sound and ML approach in line with results of RQ1 and RQ2. We recall that a sound approach (see Definition 3) takes the output of the oracle and built constraints out of "faulty" configurations/videos. It means that constraints will be very specific to the configurations given to the oracle, involving every feature and attributes with their values.

[3]Strictly speaking, configurations are invalid (resp. valid) in VM'. We use the term "faulty" (resp. "non-faulty") for keeping an unified terminology with the oracle.

Meaningfulness of constraints. A consequence is that constraints are typically difficult to read with a case-by-case extraction. A sound approach would have produced the conjunction of 53 constraints, each constraint being constituted by the number of features/attributes' values of a configuration. As a configuration corresponds to 80+ values in our case, experts would have severe difficulties to review and understand what are the precise features and attributes involved in the fault. Our proposed approach computed 5 constraints on average with only a few features/attributes. This drastic reduction in size helps a video expert to better understand the constraints.

One can argue that techniques for reducing constraints (*e.g.,* [35]) can be adapted to numerical values and perhaps improve a sound approach. However, it is unlikely since the configurations do not necessarily share common values, especially in continuous domains. In fact, there is a more fundamental issue: constraints of a sound approach are so precise they cannot be able to capture other faulty configurations even in their close neighborhood. Hence, the value of an ML approach resides in the ability to produce more general and thus meaningful constraints. The fact that constraints are general comes from the way ML algorithms are designed. They try to infer properties out of few examples resulting in an approximation of the real behaviors of data. This approximation can be seen as a convex hull in the space of configuration. The added value of ML algorithms is to allow asperities in the hull to reduce the number of classification errors that could be made by a simple convex hull approximation algorithm.

Precision and recall. The strict strategy of a sound approach has another practical implication. In our case, the sound approach would only remove 53 configurations out of 500 (no more no less). On the other hand, our ML method removes 234 more faulty configurations than a sound approach (see Results in Table 1). Indeed, when validating our classification models with 4000 new configurations, the ML approach was able to recognize 234 (as a mean over 20 runs according to Table 1) configurations as faulty – without having to produce and test the video variants. The sound approach was unable to detect them because these 234 configurations were simply not in the original sample. Hence, the prediction of faulty configurations with ML gives a factor improvement of $(48 + 234)/53 = 5,3$. (The number of videos classified as faulty during the learning process is 48 since we obtained 90.56% of classification performance. 234 corresponds to the average number of videos classified as faulty while validating the built decision tree on 4000 new configurations. Finally, 53 is the number of faulty videos in the sampling and thus classified as such by the sound approach.)

In order to scale (*i.e.,* capture a similar set of faulty configurations than an ML approach), a sound approach has to sample a significant number of additional configurations. In our case study, there are two major drawbacks. First, covering a large percentage of the configuration space is simply not possible, mainly due to numerical values. Second, the cost of testing a configuration is very high: half an hour to generate a video and a few seconds to process it with the oracle. Concretely, the cost in time and resources for 4000 configurations is $4000 * 30 = 12000 \approx 2000$ hours for one machine. Hence, the use of a sound approach can be very costly since we envision to generate even more videos in the future. Overall, the major strength of ML resides in its ability to reduce the testing cost through the prediction of faulty configurations.

The prediction capability of ML is also a weakness since it induces some errors. In our case, out of 4000 (see Table 1), in average 141 configurations were classified as non-faulty by ML (despite being actually faulty). A sound approach is also unable to forbid such configurations in the first place since they are not part of the sample. That is, a sound approach would have suffered from similar issues. Finally 69.5 configurations out of 4000 were, in average, classified by ML as faulty (despite being actually non-faulty). In this case, a sound approach would have kept these configurations and, thus, is superior to a ML technique.

As a summary there is a trade-off to find. On the one hand, the ability of ML to be one step ahead can reduce testing effort, produce meaningful constraints, and forbid more faulty configurations (as in our case study). On the other hand, a sound approach can be of interest in cases product line practitioners do not want to unnecessarily lose some configurations.

4.3 Threats to validity

External validity. There are conditions that limit our ability to generalize the results. A first threat is that we only used one product line. Thus, the results of our experiment might not be extensible to other product lines. We selected an industrial product line that has been used for more than two years now and that has passed several testing phases. We consider the variability model as realistic since numerous experts were involved in its design and there is a concrete connection with a product line. The product line generates data (videos) that differ from more traditional artifacts like code or models. However the implementation follows general principles of product lines with a variability model and a software derivation engine.

The proposed method relies on supervised ML techniques which assume that oracles are available. In our case we have to develop a specific oracle based a computer vision algorithm. In other domains, oracles might be harder to develop and only able to catch a few faulty configurations. On the other hand, traditional oracles relying on compilers or test suites can be used as well.

Internal validity. Since our earliest efforts [1, 12], the variability model has been subject to several iterations mainly due to the evolution of configuration files handled by the Lua code. Before applying the method, we considered the model as stable. Yet the model may exhibit errors (*e.g.,* wrong domains for the attributes). As part of our testing experiments, we did not observe such inconsistency of configurations' values. In fact, it could be the role of our method to detect such errors (if any).

A threat concerns the sampling of valid and faulty configurations. We trained our ML algorithm with 500 videos and then verified the learnt constraint in front of 4000 videos. To enhance internal validity, we run the experiment of learning and validating results 20 times.

One could argue that it could have been better to compare inferred constraints with existing ones. However we do not have any ground truth to rely on. We rather place ourselves in a realistic situation: inferring constraints not already specified or simply a priori unknown by developers and experts. Thus, we measure whether added constraints

help to narrow the space of possible configuration w.r.t. a set of valid products (i.e., in our case: video variants that are not blurry nor noisy). On the qualitative side, we also seek to understand whether constraints make sense for a video expert.

Another threat is that the oracle can be badly implemented, leading to incorrect judgement of the quality of some videos. First it should be noted that our ML method does not seek to undermine what faulty means; we rather learn from a set of authoritative decisions imposed by the oracle. Hence the precision and recall of our ML method (see RQ2) is not affected. However an incorrect oracle can be problematic for the expert point of view and induces a threat to RQ1. When implementing the oracle, we review a sample of dozens of faulty and non-faulty videos. Yet we cannot review all videos and there is a risk that the oracle is still not correct. What is reassuring is that the expert considered that constraints are in line with the oracle, hence giving confidence on the quality of the oracle. In general a co-design of oracle and constraints seems needed in case there are some uncertainties in the oracle's implementation.

5. DISCUSSIONS

Based on our experience, we now highlight several points that could improve the results of our general method.

Sampling techniques. In our case study, we used a fairly simple sampling technique that relies on randomly picking values of features/attributes. There are other sampling techniques and strategies to address various kinds of problems. In validation & verification, for instance, T-wise sampling is used to produce interactions between T activated features [12, 16, 23]. The assumption is that a T-wise criterion can increase the ability to detect fault. On the other hand, our basic strategy allows us to capture some diversity and to explore different areas in the configuration space. Sophisticated methods for *uniform* generation of configurations [7] can even be considered, but deserve to scale for our cases in which we have numerous numerical options. Another possible direction is to use expert knowledge to eventually guide the sampling. The sampling can be done iteratively as well.

In general, other sampling methods could allow getting fewer configurations while having a good configuration space coverage and a good intuition over the oracle's distribution. Ensuring such properties can, in the end, increase the classification performance of the ML algorithm.

Definition of oracles. We reused a method proposed in [11] to assess the quality of images and a video in terms of distribution's frequencies in the Fourier domain. The oracle is a non-trivial software procedure that may fail to detect some videos as faulty. One way to improve our oracle is to use humans (typically computer vision experts) for better reasoning about perceptual details. The counterpart is that reviewing videos can be time-consuming and even error-prone due to fatigue. Our oracle has the advantage of being an automated procedure so that we can consider the analysis of much more videos. A possible solution is to combine different oracles for mitigating the limits of some oracles. As a summary, the choice of an oracle depends on (1) the quality of its judgement and (2) its cost. Both factors can have an impact on our learning phase and perhaps be in conflict (*e.g.,* good quality but very high cost). It should be noted, however, that the selection of an oracle can be much simpler in other settings, *i.e.,* there is no tradeoff to find. For instance, a compiler is usually a fast and reliable procedure that can serve as an oracle for testing family of programs.

Machine Learning algorithm. One could wonder: why using decision trees and not other techniques? Or even, can other ML algorithms do better regarding classification performances? We chose to use decision trees as we knew the output would be very simple to understand compared to other techniques such as artificial neural network or support vector machines. Moreover, decision trees are built according to the following rule: features exposing more information entropy should be placed in the higher levels of the tree which ease the readability and understandability of extracted constraints. Nonetheless, using other ML techniques might be worth it since they could be more discriminating, thus exposing more constraints but with higher precision (*i.e.,* without introducing errors). Although, we reported that our decision tree makes classification errors, in our case, misclassifying non-faulty configurations as faulty (false positives) is not that problematic since we can produce other videos. We are aware that, in another context, false positives can be more problematic. Overall different ML strategies can be employed to either increase precision or recall.

Readability of constraints. Computer vision experts highlighted the fact that some features appear in constraints whereas they should not be discriminant in the decision of valid/non-valid configurations – for instance, the feature "vehicle1.identifier" in the constraint of Figure 7. This clearly shows a need to reduce the numbers of features taken into account when generating constraints. By lowering the number of features in the representation of data, the impact of the so-called "curse of dimensionality" will be reduced and thus, it could help building more concise and meaningful constraints. Domain knowledge can be explicitly employed for removing unnecessary features. The sampling of additional configurations is yet another possibility to further refine constraints. From this respect, an expert can incrementally guide the sampling strategy, based on her understanding of constraints.

6. RELATED WORK

Mining variability and constraints. Numerous works address the problem of synthesizing a variability model [4–6, 9, 28]. Given a Boolean formula or a set of configurations, a procedure automatically produces a variability model, including cross-tree constraints. The resulting models tend to represent the exact set of configurations, in line with the formula. However, there are cases in which the set of constraints and valid configurations is incomplete or uncertain. For instance, Nadi *et al.* showed that a large portion of configuration constraints in the Linux kernel originates from domain knowledge or necessitates testing [24]. In our case study, we report similar observations and testing needs for extracting constraints (see Section 3.1).

Perrouin *et al.* [17] proposed an automated search-based process to test and fix feature models. Preliminary results on the Linux kernel are reported. In our case, we are mostly interested in discovering new constraints rather than validating or refuting existing ones. A significant difference is that our strategy of adding constraints significantly differs. Instead of simply negating faulty configurations, we rely on machine learning to discover and capture more general faulty

configurations.

Some works address the synthesis of constraints with the use of data mining techniques. In [9], Davril *et al.* present an automated approach to extract feature models from informal, textual product descriptions. In [36], Yi *et al.* applied support vector machine and genetic techniques to mine binary constraints (requires and excludes) from Wikipedia. These works are not applicable in our context since i) we have to learn from a set of faulty configurations (not from textual content); ii) we have to deal with constraints among numerical values. Bécan *et al.* [6] targeted the problem of synthesizing attributed feature models, including constraints among attributes and features. We also produce such kinds of constraints, but we do not assume that the original set of valid configurations is sound and complete.

To the best of our knowledge, no approach infers Boolean or numerical constraints from a sample set of faulty configurations.

Prediction and product lines. The prediction of the performance of individual variants is subject to intensive research. Approaches usually handle a small sample of measured variants and seek to understand the correlation between configurations and performance [15,27,30,34,37]. The effectiveness of statistical learning techniques and regression methods have been empirically studied. Siegmund *et al.* [29] combined machine-learning and sampling heuristics to compute the individual influences of configuration options and their interactions. The approach has applications in performance-bug detection or configuration optimization. Our method also relies on sampling and learning techniques. Nevertheless, our goal differs. We aim to enforce a product line and narrow the space of possible configurations with the inference of constraints.

Verifying product lines. Numerous techniques have been developed to verify efficiently a product line based on testing, type checking, model checking, or theorem proving [32]. Testing configurable systems and product lines are an active area of research. Prior work has considered the problem of optimizing the execution of tests for a set of related products [19, 25]. Our method can benefit from test execution optimization since computations with the oracle can be done quickly.

Another research direction focuses on the identification of relevant products to test [10, 20, 21, 31, 32]. Kim *et al.* [20] applied static program analysis techniques to find irrelevant features for a test. SPLat is a dynamic analysis technique for pruning irrelevant configurations [21]. The goal is to reduce the combinatorial number of variants (e.g., Java programs) to examine. SPLif aims to detect bugs of software product lines with incomplete feature models [31]. Our method to enforce the feature models with constraints can benefit to SPLif. On the other hand, SPLif can help developers (e.g., of the video generator) prioritize failing tests and configurations for inspection.

Several techniques can be used to sample the configurations to test [3, 8, 12, 16, 18, 22, 23, 27]. In our case study, we have to deal with numerical values when sampling. We use a random strategy for picking values (see Section 3.2). We also discuss threats and possible alternatives (e.g., see [29]). Empirical studies show that sampling strategies can influence the number of detected faults or the precision of performance prediction [23,27]. For instance, increasing the size of sample sets can have positive effects on the fault-detection capability, but it also has a cost [23]. As discussed, the problem of inferring constraints is a tradeoff between the costs of testing and the ability to learn constraints from an oracle and configurations. Overall, strategies or techniques to sample or prioritize configurations to test can benefit to our method. A research direction is to conduct further empirical studies to measure their cost-effectiveness.

Random testing. Numerous black-box or white-box software testing technique have been developed [13]. The idea is to generate random inputs and resulting outputs are observed typically for detecting failures. Whitebox fuzzing, such as SAGE, consists of executing the program and gathering constraints on inputs [14]. The collected constraints are then systematically negated and solved with a constraint solver, whose solutions are mapped to new inputs. This process is repeated using search techniques and heuristics. The objective of our method differs. SAGE aims to cover as much as possible paths to detect eventually faults in single programs. Our goal is to enforce the configuration space of product lines. Hence, the role of constraints and the method to infer differ. In our case, constraints are reinjected into a variability model. SAGE exploits constraints to guide the test generation.

7. CONCLUSION

We addressed the problem of inferring constraints of a large, complex variability model in order to restrict the space of possible configurations. We proposed a method based on sampling, testing, and machine learning to identify which combinations of features/attributes (and their values) are likely to produce faulty products. From the separating functions produced by machine learning, constraints can be automatically extracted and injected into the variability model, typically to enforce a product line.

We instantiated our method on a video generator developed in the industry that originally produced irrelevant videos. Results showed that our method can classify with a good precision and recall products (videos) that were never derived and tested before. Furthermore extracted constraints express some interactions between features while remaining readable and general enough. Thanks to constraints, we can now consider the synthesis of very large datasets of truly relevant videos.

We believe the method is general enough to be applicable to product lines in other contexts and domains. For instance, a learning-based approach has the potential to infer constraints of the Linux kernel; but is it an effective solution? Further empirical studies are needed to understand the effects of sampling strategies, oracle definition, and learning process on the ability to infer constraints. More generally we invite researchers to investigate the conditions for which our method is effective or not.

8. REFERENCES

[1] M. Acher, M. Alferez, J. A. Galindo, P. Romenteau, and B. Baudry. Vivid: A variability-based tool for synthesizing video sequences. In *SPLC'14*.

[2] M. Acher, P. Collet, P. Lahire, and R. France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP)*, 78(6):657–681, 2013.

[3] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake. Similarity-based prioritization in software product-line testing. In *SPLC'14*.

[4] N. Andersen, K. Czarnecki, S. She, and A. Wasowski. Efficient synthesis of feature models. In *Proceedings of SPLC'12*.

[5] G. Bécan, M. Acher, B. Baudry, and S. Ben Nasr. Breathing ontological knowledge into feature model synthesis: an empirical study. *Empirical Software Engineering*, 2015.

[6] G. Bécan, R. Behjati, A. Gotlieb, and M. Acher. Synthesis of attributed feature models from product descriptions. In *SPLC'15*, jul 2015.

[7] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi. On parallel scalable uniform SAT witness generation. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS'15*.

[8] M. B. Cohen, M. B. Dwyer, and I. C. Society. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints : A Greedy Approach. 2008.

[9] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans. Feature model extraction from large collections of informal product descriptions. In *ESEC/FSE*, 2013.

[10] I. do Carmo Machado, J. D. McGregor, and E. Santana de Almeida. Strategies for testing products in software product lines. *ACM SIGSOFT Software Engineering Notes*, 2012.

[11] R. W. Dosselman and X. D. Yang. No-reference noise and blur detection via the fourier transform. Technical report, University of Regina, CANADA, 2012.

[12] J. A. Galindo, M. Alférez, M. Acher, B. Baudry, and D. Benavides. A variability-based testing approach for synthesizing video sequences. In *International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 293–303. ACM, 2014.

[13] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 2005.

[14] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 2012.

[15] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *ASE*, 2013.

[16] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Trans. Software Eng.*, 2014.

[17] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. Le Traon. Towards automated testing and fixing of re-engineered feature models. In *ICSE '13 (NIER track)*, 2013.

[18] M. F. Johansen, O. y. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. *SPLC'12*, 2012.

[19] C. Kim, S. Khurshid, and D. Batory. Shared execution for efficiently testing product lines. In *Software Reliability Engineering (ISSRE), 2012*.

[20] C. H. P. Kim, D. S. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *AOSD'11*.

[21] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. D.Amorim. Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *ESEC/FSE 2013*, 2013.

[22] B. P. Lamancha and M. P. Usaola. Testing Product Generation in Software Product Lines. 2010.

[23] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. A comparison of 10 sampling algorithms for configurable systems. In *ICSE'16*.

[24] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Where do configuration constraints stem from? an extraction approach and an empirical study. *IEEE Trans. Software Eng.*

[25] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *ICSE'14*.

[26] C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.

[27] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. Cost-efficient sampling for performance prediction of configurable systems (t). In *ASE'15*.

[28] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *ICSE'11*.

[29] N. Siegmund, A. Grebhahn, C. Kästner, and S. Apel. Performance-influence models for highly configurable systems. In *ESEC/FSE'15*.

[30] N. Siegmund, M. RosenmüLler, C. KäStner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Inf. Softw. Technol.*, 2013.

[31] S. Souto, D. Gopinath, M. d'Amorim, D. Marinov, S. Khurshid, and D. Batory. Faster bug detection for software product lines with incomplete feature models. In *SPLC '15*.

[32] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 2014.

[33] T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *ICSE'09*, pages 254–264. ACM, 2009.

[34] P. Valov, J. Guo, and K. Czarnecki. Empirical comparison of regression methods for variability-aware performance prediction. In *SPLC'15*.

[35] A. von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger. Presence-condition simplification in highly configurable systems. In *ICSE*, 2015.

[36] L. Yi, W. Zhang, H. Zhao, Z. Jin, and H. Mei. Mining binary constraints in the construction of feature models. In *RE'12*, 2012.

[37] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki. Performance prediction of configurable software systems by fourier learning (T). In *ASE'15*.